Initiation aux langages de commandes et à la programmation sous Unix

DESS M3I (année 2004-2005)

Luc Mieussens
mieussens@mip.ups-tlse.fr
laboratoire MIP
Université Paul Sabatier - Toulouse 3

Table des matières

1 Le système unix/linux

- unix = système d'exploitation d'un ordinateur :
  → gestion des processus (exécution de programmes)
  → gestion des fichiers (données)
  → gestion des périphériques et de réseaux

- particularités (ordinateur mono-processeur sous unix) :
  → multi-utilisateurs
  → multi-taches
  → temps partagé

- extension aux ordinateurs multi-processeurs

– historique :

→ années 70 : mise au point de la première version chez AT&T Bell Labs, puis à l' U.C. Berkeley

→ années 80 :

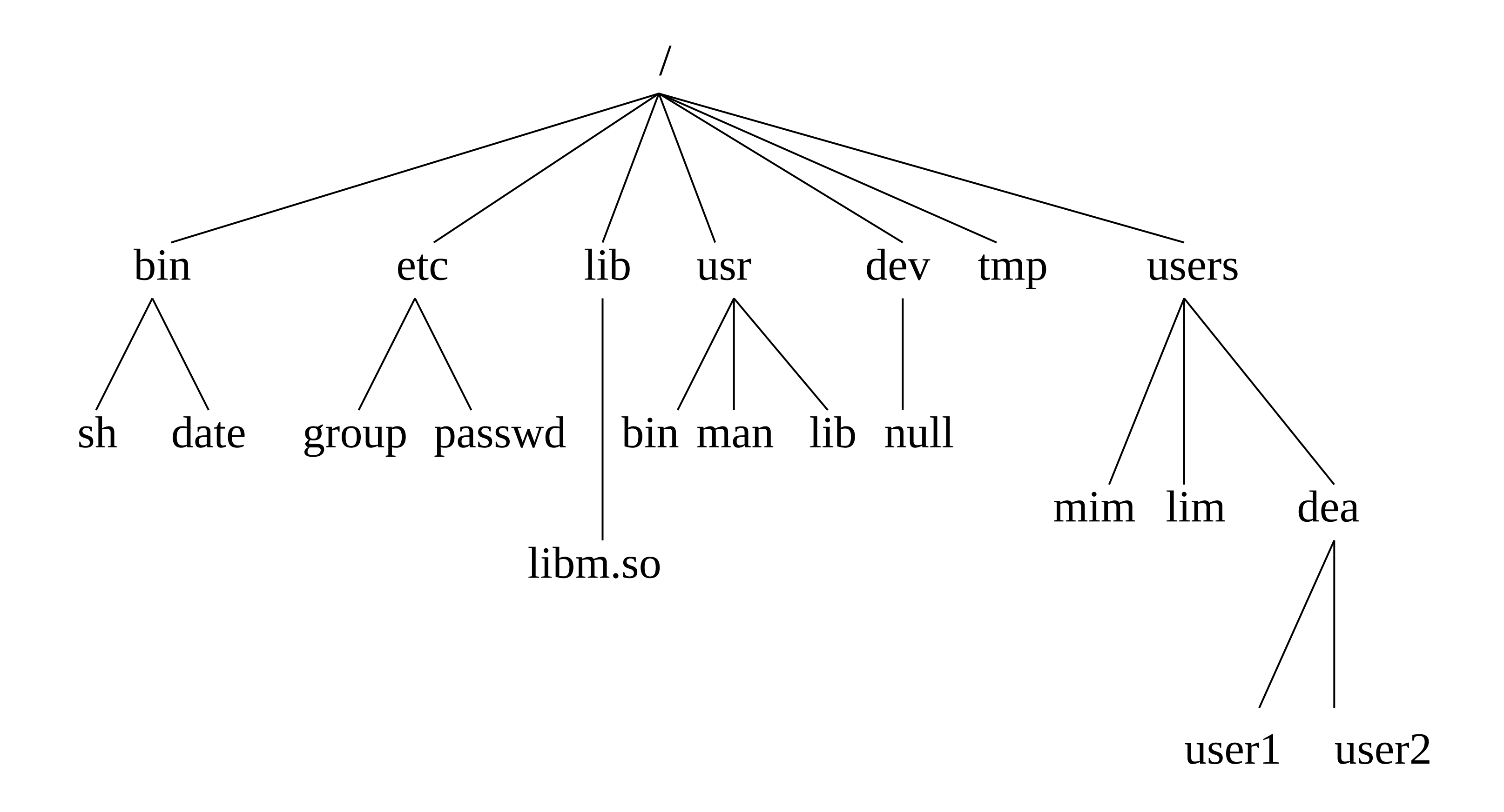
- DOS reprend des idées d'unix
- versions commerciales d'unix
- création de la FSF et du projet GNU

→ années 90 : mise au point de linux

2 Le système de fichiers

- fichier : ensemble de données, stockées sous forme de caractères dans la mémoire de l'ordinateur
- caractéristiques :
  - type (ordinaire, répertoire)
  - taille
  - identité du propriétaire (UID-GID)
  - droits d'accès en lecture/écriture/exécution
  - dates de modifications

– organisation en arborescence :



- les chemins d'accès :
  - chemin relatif (au répertoire courant)

    mot/mot/mot/...

    où un mot est le nom d'un sous-répertoire, ou un des caractères spéciaux suivants :

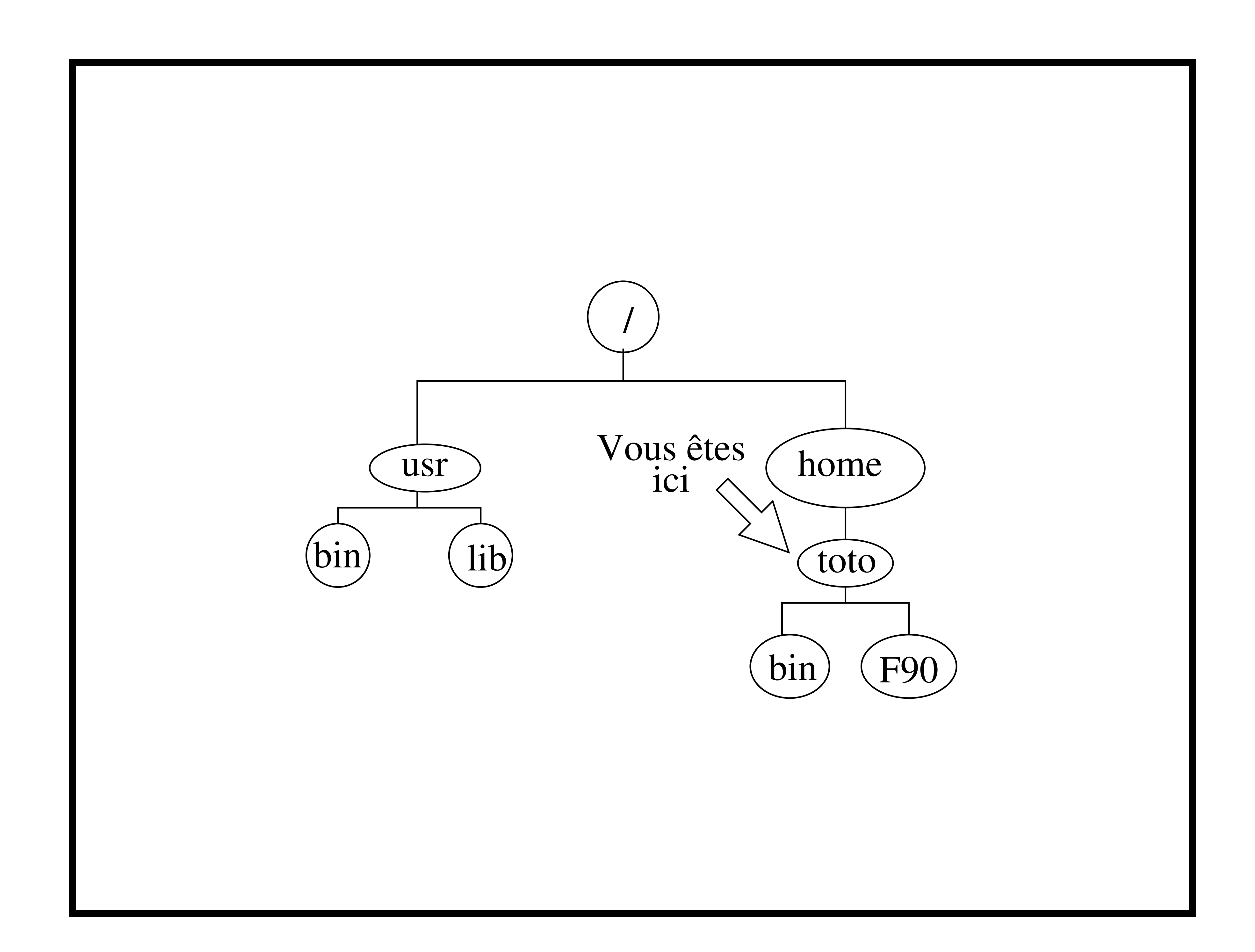
.	le répertoire courant
..	le répertoire père
~	le répertoire "home"

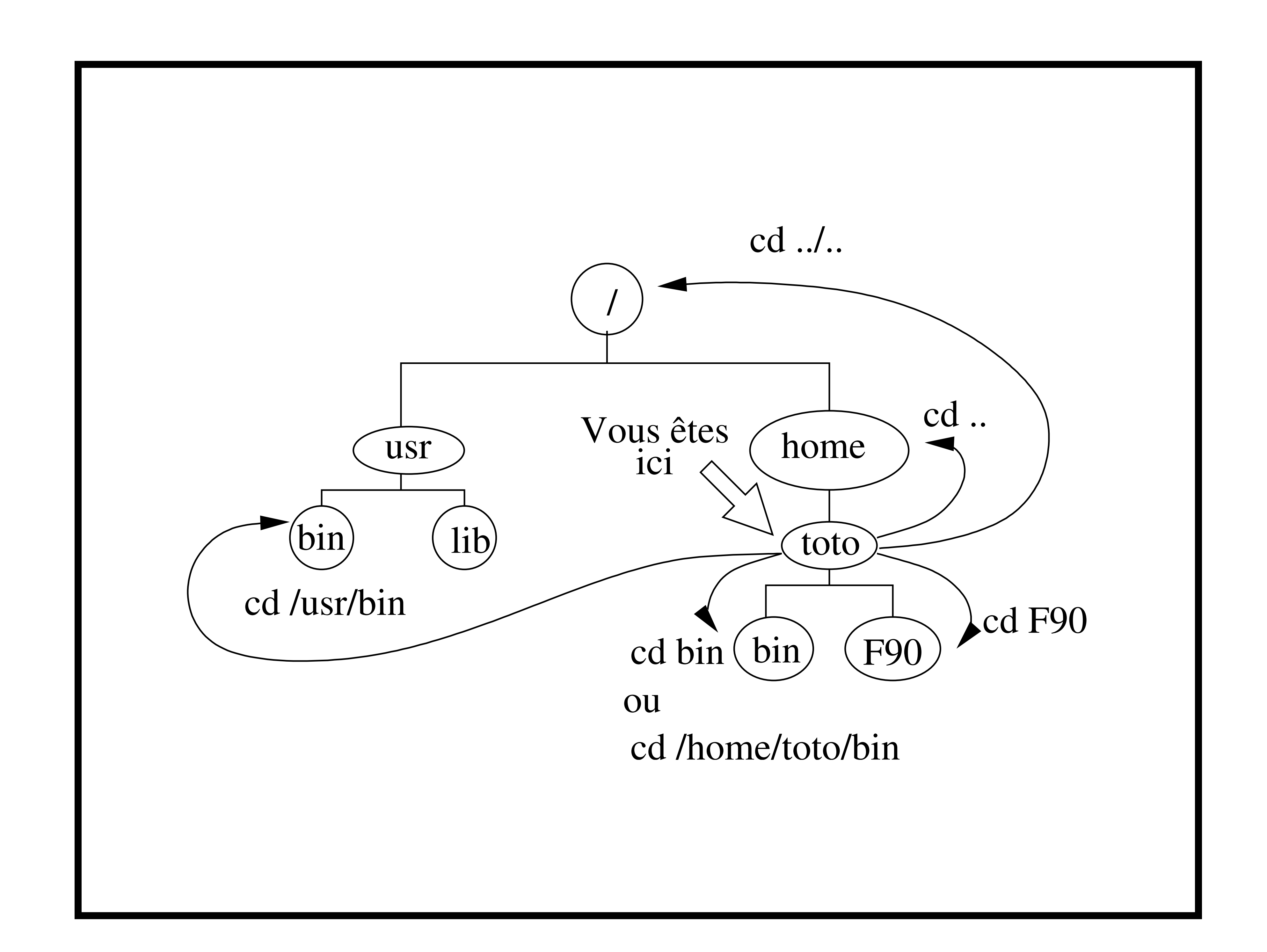
  - chemin absolu : on part de la racine /

    /mot/mot/mot/...

– Navigation dans les répertoires

`pwd`	affiche le répertoire de travail
`cd rep`	déplacement dans le répertoire rep
`mkdir repertoire`	crée le répertoire
`rmdir repertoire`	efface un répertoire


/
usr
bin
lib
Vous êtes
ici
home
toto
bin
F90

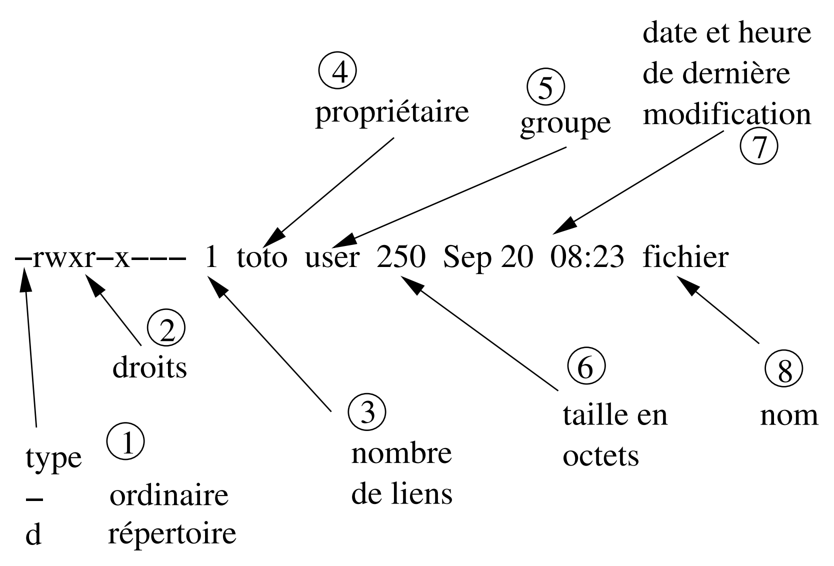

cd ../..
/
usr
home
cd ..
Vous êtes
ici
bin
lib
toto
cd /usr/bin
cd F90
cd bin
bin
F90
ou
cd /home/toto/bin

– Lister le contenu d'un répertoire

`ls` *[options] [arguments]*

-a	affiche tous les fichiers
-d	affiche le répertoire, pas son contenu
-l	affichage long (liens, dates, propriétaire, taille)

le résultat de la commande `ls -l` contient pour chaque fichier une ligne du type :



– droits d'accès désignées par :

r	droit de lecture
w	droit d'écriture
x	droit d'exécution
-	pas de droit

Exemple :

`-rw-r--r-- 1 licm1 users 1234 Mar 6 11 :24 toto.txt`

– gestion des droits d'accès

`chmod`	change les droits d'un fichier ou d'un répertoire
`chgrp`	change le groupe d'un fichier
`chown`	change le propriétaire d'un fichier

– changer les droits d'accès d'un fichier

$$\texttt{chmod}\ \begin{bmatrix} u \\ o \\ g \end{bmatrix} \begin{bmatrix} + \\ - \end{bmatrix} \begin{bmatrix} r \\ w \\ x \end{bmatrix}\ \texttt{fichier}$$

u utilisateur propriétaire du fichier

g utilisateurs du groupe

o les autres

$+$ ajoute les droits qui suivent

$-$ enlève les droits qui suivent

exemple :

`chmod u+w file`	donne au propriétaire le droit d'écriture
`chmod ug+rx file`	donne au propriétaire et au groupe les droits de lecture et d'exécution
`chmod o+r-x file`	donne aux autres le droit de lecture et enlève le droit d'exécution

– maintenance de fichiers

`rm fichier`	efface le fichier
`cp fich1 fich2`	copie fich1 dans fich2
`mv fich1 fich2`	déplace/renomme fich1 en fich2

– commandes diverses

`echo`	affiche une chaîne de caractères à l'écran
`cat`	affiche le contenu d'un fichier sur l'écran et permet de concaténer des fichiers
`head, tail`	affiche les 10 premières (resp. dernières) lignes d'un fichier sur l'écran
`more`, `less`	affiche un fichier page par page
`basename, dirname`	extrait une partie du nom d'un fichier
`sort`	trie les lignes d'un fichier dans l'ordre lexic.
`wc`	compte le nombre de lignes d'un fichier
`touch`	actualise les dates d'un fichier (créé si inexistant)
`find`	trouver un fichier dans l'arborescence
`diff`	comparer le contenu de deux fichiers

3 Le langage de commandes Shell

3.1 Introduction

- logiciel qui s'utilise comme un langage qui sert d'interface entre unix et l'utilisateur
- forme un ensemble de commandes qui s'ajoutent aux commandes de base unix
- ces commandes travaillent en général sur des fichiers
- permet d'écrire des programmes utilisant les commandes de base unix et les commandes supplémentaires
- chaque commande est un fichier exécutable : pas de compilation
- langage interprété : chaque ligne est analysée puis immédiatement exécutée

- deux utilisation possibles :

  - mode interactif : on écrit une ligne, on la valide (entrée), elle est exécutée

  - mode programme (script) : plusieurs lignes sont stockées dans un fichier texte et traitées séquentiellement.

    exemple :

    `ls -l prog.*`

    ligne analysée : option :`-l`, argument d'entrée : tous le fichiers dont le nom commence par `prog.`, ceci est fourni à la commande `ls`.

- langage algorithmique : structures de contrôle (si, tant que), boucles (pour), utilisation de variables, manipulation de fichiers et de répertoires

- différents shell, exemples :
  - **`sh`** : "Bourne shell" shell originel, présent sur tous les systèmes
  - **`ksh`** : "Korn shell" plus complet, pas présent partout
  - **`bash`** : "Bourne again shell" shell utilisé sous Linux, contient l'essentiel de **`sh`** et beaucoup des caractéristiques de **`ksh`**.

  attention : sur ondine le shell par défaut est **`csh`**, *incompatible avec les autres (utiliser* **`ksh`**, *ou bien* **`sh`** *en imposant* **`_XPG=1`**).

- syntaxe d'une ligne de commande :

```
commande argument1 argument2 ... argumentn
```

- POUR OBTENIR DE L'AIDE SUR UNE COMMANDE SHELL OU UNIX

  → manuel : `man` commande

  → documentation texinfo : `info` commande

- POUR OBTENIR DE L'AIDE SUR LE SHELL

  → manuel : `man` shell

– interprétation d'une ligne de commande : elle se fait en plusieurs étapes, effectuées dans un ordre précis :

(a) interpéter les séparateurs et séparer les champs

(b) isoler les chaînes de caractères encadrées par des quotes '

(c.1) remplacer les variables par leur contenu

(c.2) exécuter les commandes encadrées par des anti-quotes ` ou la séquence `$()` et les remplacer par leur résultat

(c.3) évaluer les expressions arithmétiques

(d) isoler les chaînes de caractères encadrées par des doubles quotes "

(e) remplacer les caractères spéciaux (`*` `[]` `?` `~` etc.) par leur valeur

(f) repérer les séquences `()` et `{}`

(g) positionner certains paramètres en fonction des caractères spéciaux `|` `&` `&&` `||`

(h) mettre en place les redirections

(i) localiser les commandes élémentaires (alias, fonctions path)

3.2 Séparateurs

- séparateur dans une ligne : espace (noté SPC dans le cours) et tabulation (noté TAB)
- conséquence : deux chaînes accolées forme une seule chaîne (concaténation)
- liste des séparateurs stockées dans la variable `IFS`. Cette liste est modifiable dans une macro-commande (cf 3.9)

3.3 Caractère spécial quote '

- les chaînes encadrées `'chaîne'` ne sont pas interprétées. Autrement dit, aucun caractère spécial n'a de signification, même un séparateur.
- exemple :

```
% echo $SHELL
/bin/sh
% echo '$SHELL'
$SHELL
```

3.4 Les variables

- nom de variable : chaîne composée de lettres (a-z,A-Z), de chiffres (0-9) et de souligné _
- valeur d'une variable : chaîne de caractères
- affectation : `var=chaîne`

  on dit que la variable **`var`** est positionnée à la valeur **`chaîne`**
- initialisation : `var=`
- substitution d'une variable par sa valeur : `$var`

  on parle alors d'expansion de la variable **`var`**.
- expansion conditionnelle : si var est initialisée et non vide, on la substitue par sa valeur, sinon elle est remplacée par la chaîne mot (autres expansions : voir man sh)

  `${var:-mot}`
- tableaux de variables : possible en **`bash`**, pas en **`sh`**

exemple 1 : affectation/expansion

```
% toto=recu
% titi='cinq sur 5'
% tutu=$toto
% echo toto
toto
% echo $toto
recu
% echo $tutu $titi
recu cinq sur 5
```

exemple 2 : concaténation

```
% toto=bon
% titi=jour
```

```
% tutu=$toto$titi
% echo $tutu
bonjour
```

exemple 3 : concaténation robuste : on isole la variable entre {}, précédée de `$`

```
% toto=bon
% echo $totojour

% echo ${toto}jour
bonjour
```

exemple 4 : expansion conditionnelle (utile pour tester la présence d'un paramètre dans un programme)

```
% var=
```

```
% echo ${var:-toto}
toto
% var=${var:-toto} ; echo $var
toto
% var=${var:-titi} ; echo $var
toto
```

- variables prépositionnées
  - à la naissance d'un processus shell, des variables sont définies et positionnées (variables d'environnement)
  - exemples :

`0`	nom du programme shell en cours
`1` ... `n`	les n paramètres passés au programme lors de son appel
`#`	nombre de ces paramètres
`*`	liste de ces paramètres
`$`	numéro du processus courant
`HOME`	répertoire principal de l'utilisateur
`PWD`	répertoire courant
`PS1`	invite primaire ("prompt"), modifiable (dans ce cours on suppose que PS1=%)

3.5 substitution d'une commande par son résultat : caractère spécial contre-quote ` ou `$()`

en sh : `` `chaîne` ``

en bash : `$(chaîne)`

1. la chaîne est considérée comme une commande
2. elle est exécutée
3. la séquence est remplacée par le résultat de la commande

exemple :

```
%ls
toto titi.f tutu.c
%echo $(ls)
toto titi.f tutu.c
%var=$(ls)
%echo $var
toto titi.f tutu.c
%var2=$(echo '$var')
%echo $var2
$var
```

3.6 Expressions arithmétiques

- expressions ne contenant que des opérations arithmétiques (`+ - * /`) entre variables de type <u>entier</u>
- syntaxe

  `var*n+10` ou `'(var*n +10)/2'`

  → le `$` devant les variables n'est pas nécessaire
- évaluation d'une expression arithmétique :

  `((expression))` ou `let "expression"`
- substitution d'une expression arithmétique par son résultat

  `$((expression))`
- affectation du résultat à une variable

  `var=$((expression))` ou `((var=expression))`
- en `sh`, le seul moyen est la commande `expr` d'UNIX

exemple :

```
% var=4
% echo $((2*var))
8
% echo $((12345679*27))
333333333
% var=$((5/2))
% echo $var
2
% var=`expr 2 \* $var` ; echo $var
4
```

3.7 Caractère spécial double quote "

- dans toute chaîne encadrée `"chaîne"`, les caractères spéciaux perdent leur signification
- différence avec le quote ' : à cette étape, les expansions de variables et de commandes ont déjà été effectuées
- autrement dit : tous les caractères spéciaux perdent leur signification, sauf `$`

exemples :

```
% ls
toto.f toto.o
% ls toto*
toto.f toto.o
% ls "toto*"
toto* not found
% echo "$PWD"
```

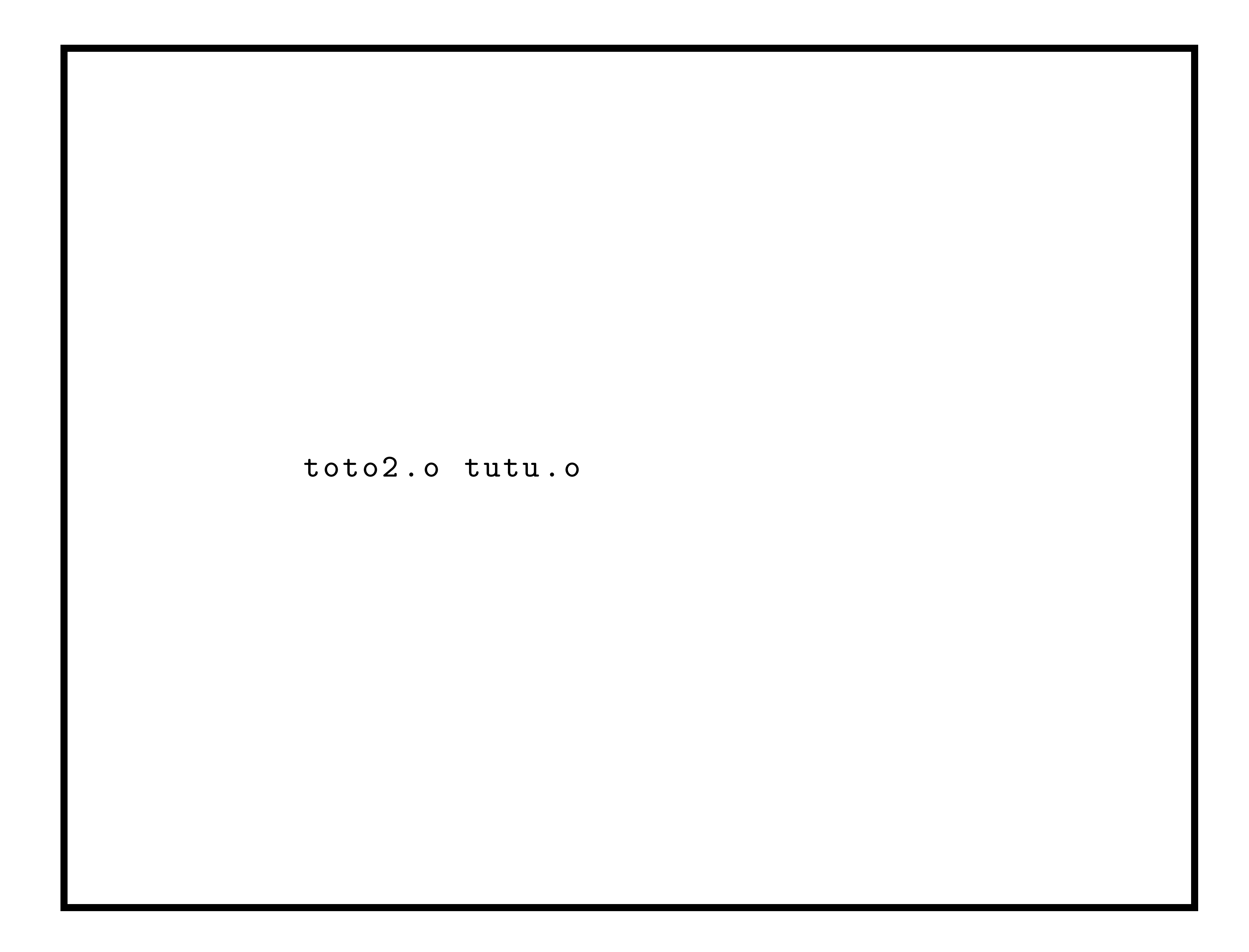

/home/user/

3.8 Autres caractères spéciaux : expressions génériques

- but : simplifier la désignation de noms de fichiers
- expression (ou motif) générique (pattern en anglais) : chaîne contenant un ou plusieurs caractères spéciaux
- le shell cherche dans le répertoire courant les fichiers dont le nom correspond à la description abrégée

*	partie ou totalité d'un nom

```
% ls
toto1.f toto2.o toto_base.f tutu.o
% ls toto*.f
toto1.f toto2.f toto_base.f
% echo *.o
```


toto2.o tutu.o

`?`	un caractère quelconque

```
% ls
toto1.f toto2.o toto_base.f tutu.o
% ls toto?.f
toto1.f toto2.f
```

`[]`	un caractère parmi ceux spécifiés

`[a3]`	a ou 3
`[1-10]`	1 ou 2 ... ou 10
`[a-e]`	a ou b ... ou e
`[aA-fF]`	a ou b ... ou f ou A ou B ... ou F
`[!1-5]`	un caractère différent de 1,2 ... ,5

```
% ls
toto1.f toto2.o toto_base.f tutu.o
% ls toto[!_].f
toto1.f
```

`~`	répertoire personnel (home)

`/`	séparateur de répertoire dans un chemin

\	rend inactif le caractère spécial qui suit

3.9 Séquences () et {}

- servent à isoler une ou un groupe de commande (appelé alors macro-commande)
- `()` : s'exécute dans un environnement séparé
- `{}` : s'exécute dans l'environnement courant

exemple :

```
% pwd
/tmp
% (cd /usr ; pwd)
/usr
% pwd
/tmp
```

```
% pwd
/tmp
% {cd /usr ; pwd}
/usr
% pwd
/usr
```

3.10 Caractères ; `|` `&` `&&`

Ils définissent la fin d'une commande élémentaire

`cmd1` $\diamond$ `cmd2`

où $\diamond$ est

`;`	séquentiel (cmd2 est lancée à la fin de cmd1)
`&`	parallèle (cmd2 est lancée avant la fin de cmd1)
	(sert aussi à lancer une commande en arrière-plan)
`|`	pipe (stdin de cmd2 est prise dans stdout de cmd1)
`&&`	cmd2 exécutée si cmd1 a réussi
`||`	cmd2 exécutée si cmd1 a échoué

Remarque : `commande &` permet de récupérer la main avant la fin de la commande

exemples :

```
% echo "liste fichiers" ; ls
liste fichiers
toto1.f toto2.o toto_base.f tutu.o
% netscape & ls
[4] 1821
toto1.f toto2.o toto_base.f tutu.o
% ls -l | wc -l
5
% ls titi && echo "fichier existe"
titi not found
% ls titi || echo "fichier inexistant"
titi not found
fichier inexistant
```

3.11 Redirections

- but : rediriger stdin, stdout ou stderr ou tout autre fichier d'entrée/sortie d'une commande **`cmd`** vers un autre fichier
- les redirections s'effectuent avec des noms de fichiers ou des descripteurs associés à des fichiers
- syntaxe :

  `cmd` *[options] [arguments]* <u>*redirections*</u> (en fin de ligne)

  où les redirections s'écrivent ainsi

- en lecture : **`<`**

`0<fich`	stdin redirigée sur le fichier fich
`0<&n`	stdin redirigée sur fichier de descripteur n

– en écriture : `>`

`1>toto 2>titi`	stdout et stderr redirigées sur les fichiers titi et toto (créés ou écrasés)
`>toto`	stdout redirigée sur toto (pas besoin du descripteur 1)
`2>&n`	stderr redirigée sur le fichier de descripteur n (créé ou écrasé)

remarquer le `&` devant le descripteur de redirection en écriture

– fichier de sortie utile :

`/dev/null`

fichier vide, tout ce qui y est envoyé est perdu

- exemples :

```
% ls
toto1.f toto2.o toto_base.f tutu.o
% ls 1>liste
% cat liste
toto1.f toto2.o toto_base.f tutu.o
% ls titi 2>erreurs
% cat erreurs
titi not found
```

- redirection en écriture en mode ajout : `>>`

`1>>fich`	stdout redirigée en écriture sur le fichier fich, créé ou complété
`1>>&n`	stdin redirigée sur fichier de descripteur n, créé ou complété

– lecture en ligne : pour une commande dont la stdin est le clavier

```
commande   [options] [arguments] << [-]mot
           ligne 1
           ligne 2
           ...
           mot
```

– les lignes de la ligne 1 à la dernière avant mot sont prises comme stdin de la commande (à la place du clavier)
– si le mot est précédé de `-` les blancs et les tabulations des débuts de ligne sont ignorés
– les lignes sont interprétées, sauf si le mot est encadré de quotes '
– utile pour lancer une commande interactive en arrière-plan

- exemple :

```
% dc << EOF
>1
>2
>+
>p
>EOF
3
```

- redirection permanente

  `exec` *redirection*

  la redirection est valide pout le shell en cours et tous les shells engendrés

  exemple :

  `exec 2>liste_erreurs`

  pour le shell en cours, toutes les erreurs sont affichées dans le fichier liste_erreurs.

3.12 Localisation de la commande

- le shell détermine si la commande est une commande élémentaire, un alias, une fonction, ou une commande dont le chemin d'accès est définie dans le path
- alias

```
alias chaîne1=chaîne2
```

  - définit le synonyme chaîne1 de la chaîne de caractère chaîne2
  - utile pour donner des noms cours à des commandes complexes
  - exemples :

```
alias ll="ls -l"
```

  - pour avoir la liste des alias : taper

```
alias
```

    suivi d'entrée
  - pour annuler l'alias chaîne1 :

```
unalias chaîne1
```

- fonction (voir 5.4)

- le path :
  - la variable `PATH` contient une liste de répertoires

    ```
    rep1:rep2 :... :repn
    ```

    dans lesquels le shell va chercher le fichier exécutable correspondant au nom d'une commande
  - la recherche est effectuée dans l'ordre de la liste
  - pour ses propres commandes, l'utilisateur peut rajouter des répertoires à cette liste en incrémentant la variable `PATH`

    ```
    PATH=$PATH:toto
    ```

4 Contrôle des processus

- liste des processus :

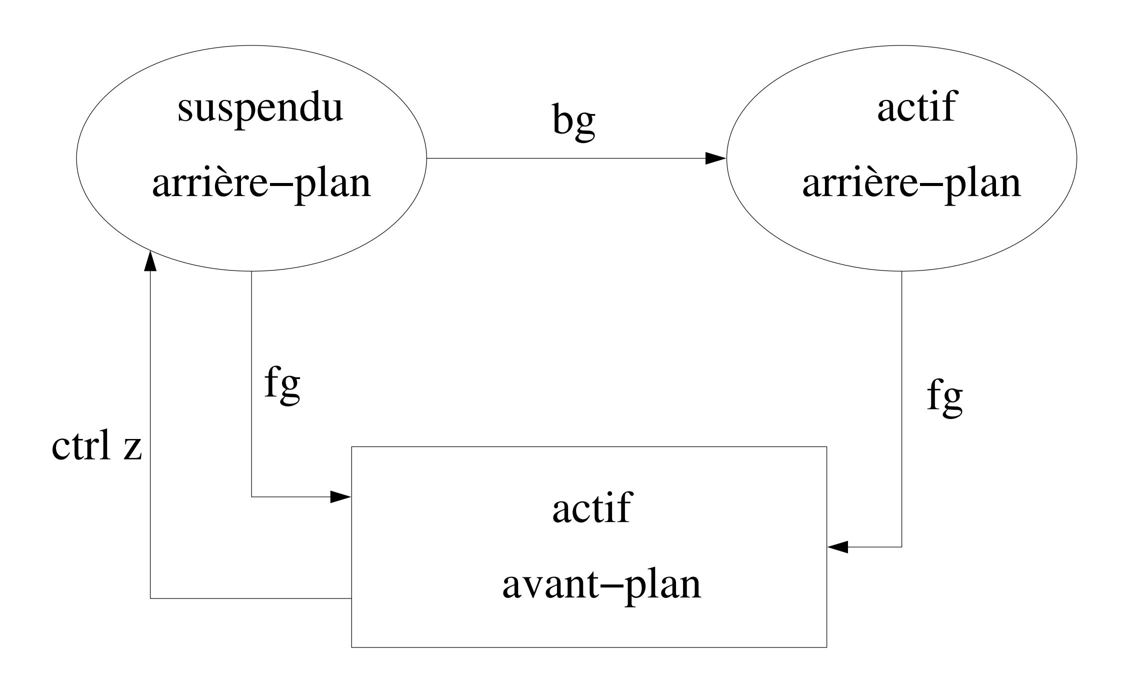
  `ps -u utilisateur`

  cette commande donne une liste d'informations sur tous les processus encore vivants, lancés par le système et l'utilisateur depuis la connexion, notamment leur numéro d'identification (`PID`)
- le `PID` est unique
- liste des processus lancés par l'utilisateur avec le shell courant

  `jobs`

  ne donne pas le `PID` mais seulement un numéro local
- le contrôle d'un processus est effectué par son PID
- un processus peut être actif ou suspendu, en avant-plan (foreground) ou en arrière-plan (background)

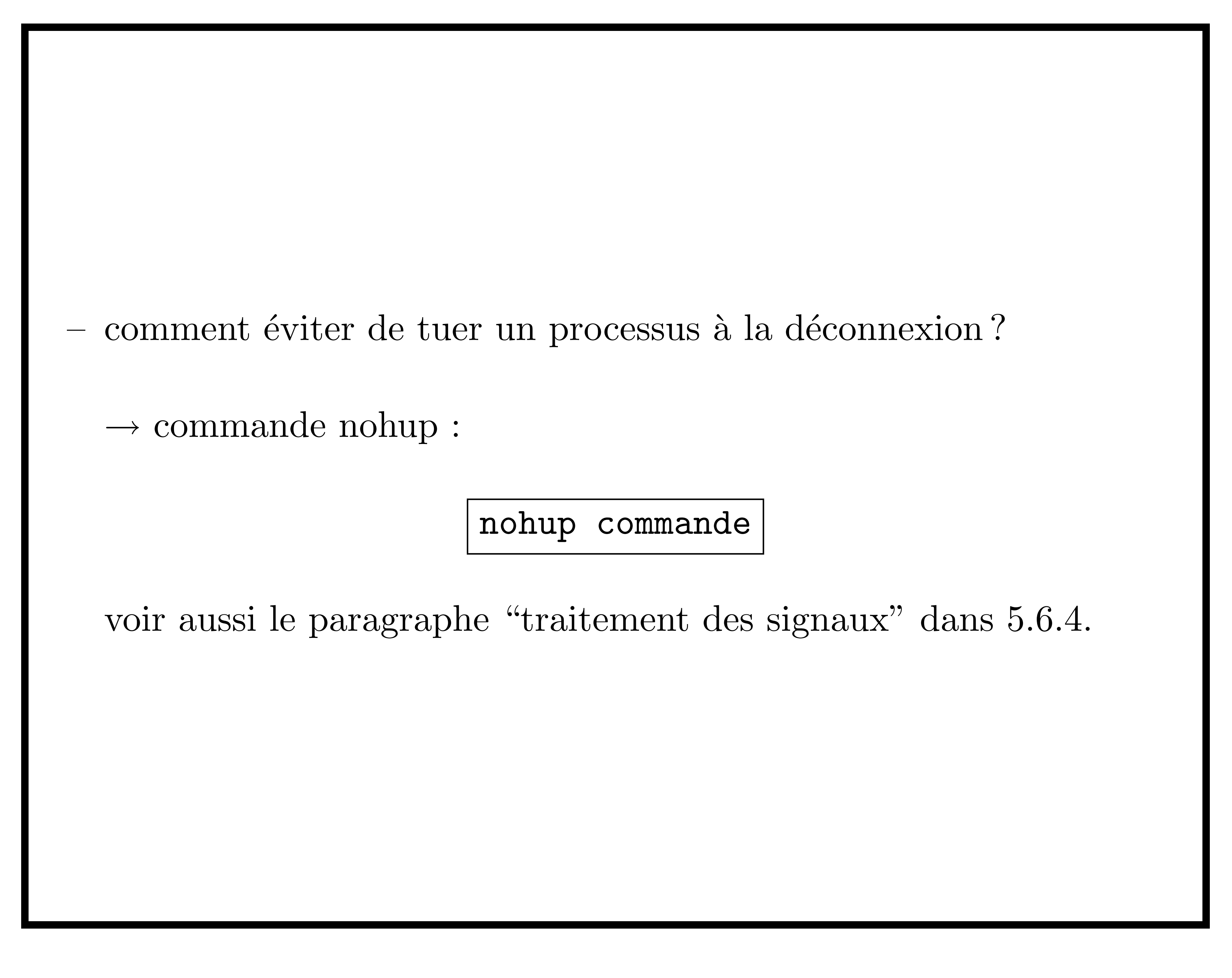
– passage d'un état à l'autre



– pour traîter un autre processus que le dernier lancé, les commandes `fg` et `bg` doivent être complétées par le numéro local du processus donné par la commande `jobs`
– contrôle plus complet des processus : voir 5.6

– exemple :

```
% netscape
     {ctrl z}
[1]+  Stopped                 netscape
% jobs
[1]+  Stopped                 netscape
% bg
[1]+ netscape &
% jobs
[1]+  Running                 netscape &
% ps
  PID TTY          TIME CMD
 1879 pts/3    00:00:00 bash
 1983 pts/3    00:00:00 netscape-commun
 1987 pts/3    00:00:00 ps
```


– comment éviter de tuer un processus à la déconnexion ?
→ commande nohup :
nohup commande
voir aussi le paragraphe “traitement des signaux” dans 5.6.4.

5 Les scripts

5.1 Introduction

- programme stocké dans un fichier contenant
  - des commandes écrites en shell
  - des fonctions
  - des commandes unix
  - des appels à d'autres scripts
  - des commentaires (caractère `#`)
- possède des paramètres (nommés `$1,$2,...`)
- possède des structures de contrôle (`if, for, while`)
- est utilisable comme une commande shell (le fichier doit avoir le droit d'exécution `x`)
- peut utiliser une autre shell que le shell courant
- le shell exécute chaque ligne comme une ligne de commande, en commençant donc par une phase d'interprétation.

pour exécuter un script `prog` :

- le fichier prog doit avoir le droit d'exécution `x`
- si le répertoire courant est dans le path, on peut utiliser le script comme une commande unix en tapant `prog`
- sinon, il faut préciser que le script se trouve dans le répertoire courant, en utilisant son chemin absolu, ou plus simplement son chemin relatif :

  `./prog`

  sinon, le système ne trouve pas le script dans le path et affiche un message d'erreur
- on peut aussi utiliser la commande `.` :

  `. ./prog`

  qui permet à prog d'éventuellement affecter les variables d'environnement. Par exemple en cas de modification du fichier `.bashrc`, on peut remettre à jour l'environnement en tapant

  `. ./bashrc`

5.2 Structures de boucles et de test

– boucle pour

```
for variable in liste
    do
      suite de commandes
    done
```

→ la suite de commandes est exécutée pour chaque valeur de la variable prise dans la liste

→ la liste peut être donnée sous forme d'expression générique

exemple :

```
for prog in *.f
    do
      echo "fichier"  $prog
      echo
      head -1 $prog
    done
```

liste la première ligne de tous les fichiers fortran avec la commande `head`.

– boucle tant que

```
while liste_commandes_1
      do
        liste_commandes_2
      done
```

1) La liste_commande_1 est exécutée
2) si la dernière commande a réussi (code de retour 0) la liste_commandes_2 est exécutée et on repasse à 1, sinon on sort de la boucle.

exemple :

```
echo "fichier a detruire :"
while  read fichier ; rm fichier
       do
         echo "autre fichier ?"
       done
echo $fichier "inexistant ou protege"
```

Tant que le fichier peut-être détruit (existant et avec les droits en écriture), il est détruit et on demande un nouveau fichier.

– test si

```
if liste_commandes_1
   then
       liste_commandes_2
   elif liste_commandes_3
        then
            liste_commandes_4
   else
       liste_commandes_5
fi
```

si la dernière commande de liste_commandes_1 réussie, alors liste_commandes_2 est exécutée, sinon etc.

en général, liste_commandes_1 est l'évaluation d'une expression conditionnelle (cf 5.3).

– test case

```
case expression in
     liste_valeurs_1) liste_commandes_1 ;;
     liste_valeurs_2) liste_commandes_2 ;;
     ...
                   *) liste_commandes_n ;;
esac
```

→ l'expression est évaluée,
→ si elle est dans liste_valeurs_1, alors liste_commandes_1 est exécutée, et on sort du `case`.
→ sinon on passe à la liste_valeurs_2, etc.
→ une liste de valeurs peut être donnée par une ou plusieurs expressions génériques, séparées par des |

La liste * correspond à n'importe quelle valeur.

exemple :

```
    echo "nom du logiciel"
    read logiciel
    case $logiciel in
           matlab)  matlab ;;
maple | xmaple )  xmaple ;;
               *)  echo "$logiciel non disponible" ;;
    esac
```

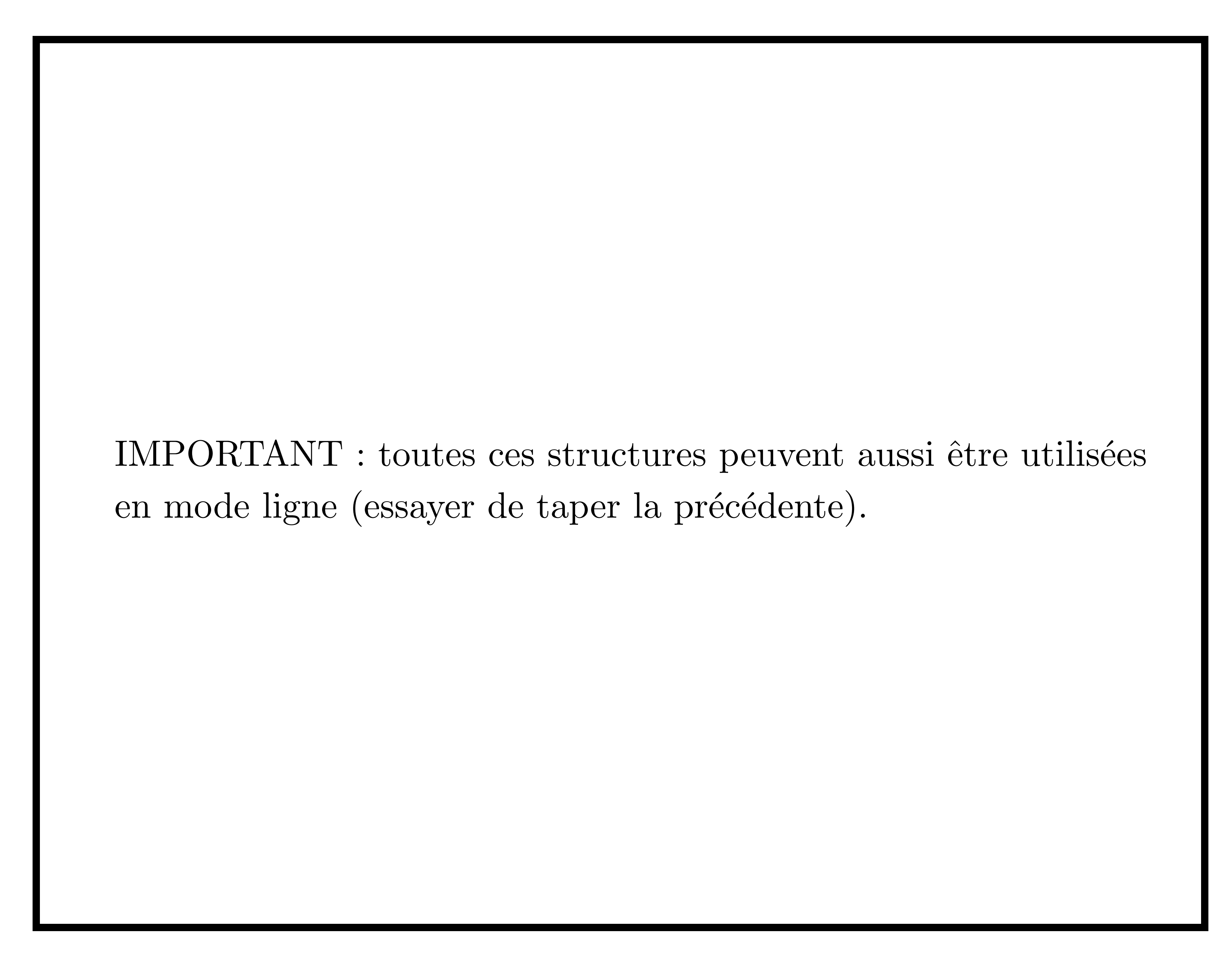
→ le nom d'un logiciel est lu au clavier (**read**),
→ si ce nom est matlab, alors matlab est lancé,
→ sinon, si ce nom est maple ou xmaple, alors xmaple est lancé
→ sinon un message est écrit à l'écran

- structures de contrôle
  - `break` sortie de boucle
  - `exit` (ou `exit n`) sortie de programme avec le code de retour de la dernière commande ou le code n.
  - `return` (ou `return n`) idem pour une fonction
  - `:` instruction vide (ne fait rien, code de retour toujours 0)
  - `. fichier` exécute les commandes du fichier sans créer de nouveau processus

exemple :

```
while  :
  do
    echo "continuer ?"
    read reponse
    case $reponse in
         [oO]|[oO][uU][iI]) echo "OK" ; break ;;
         [nN]|[nN][oO][nN]) echo "stop" ; exit ;;
    esac
  done
  echo "suite du programme"
```

→ tant que la réponse n'est ni oui ni non, la question est reposée
→ si la réponse est oui, on passe à la suite (sortie du while)
→ si c'est non, on arrête (sortie du programme)



IMPORTANT : toutes ces structures peuvent aussi être utilisées en mode ligne (essayer de taper la précédente).

5.3 Expressions conditionnelles

- expression qui après évaluation renvoie vrai (0) ou faux ($\neq$0)
- évaluation par l'opérateur `test` ou `[ ]`

  `[ expression ]` ou `test expression`

  renvoie la valeur 0 si l'expression est vraie, $\neq$ 0 si elle est fausse

– comparaison d'expressions arithmétiques

$$\texttt{expr1} \diamond \texttt{expr2}$$

avec les opérateurs suivants

$\diamond$	-eq	-ne	-gt	-lt	-ge	-le
signifie	$=$	$\neq$	$>$	$<$	$\geq$	$\leq$

exemples :

```
[ 2 -eq 3 ] &&  echo vrai || echo faux
[ $((2+1)) -eq 3 ] && echo vrai
x=2 ; [ $(((x+1)*2)) -gt 5 ] && echo vrai
```

– comparaisons entre chaînes

`-z chaîne`	vrai si chaîne de longueur nulle
`-n chaîne`	vrai si chaîne de longueur non nulle
`chaîne_1 = chaîne_2`	vrai si chaînes égales
`chaîne_1 != chaîne_2`	vrai si chaînes différentes

exemples :

```
var=toto ; [ -z $var ] || echo longueur non nulle
[ toto = $var ] && echo vrai
[ -n toto ] && echo longueur non nulle
[ -z "" ] && echo longueur nulle
[ -n  ] && echo longueur non nulle
```

attention à la confusion entre chaîne de longueur nulle et espace (utiliser des double-quotes si nécessaire)

– tests sur fichiers

`-a fich`	vrai si fich existe
`-d fich`	vrai si fich est un répertoire
`fich_1 -nt fich_2`	vrai si fich_1 plus récent que fich_2

– combinaison d'expressions conditionnelles par opérateurs booléens

`&&`	et
`||`	ou
`!`	non
`( )`	association

il faut alors utiliser l'opérateur d'évaluation `[[ ]]` au lieu de `[ ]`

exemple :

```
% [ 3 -eq 3 && (toto = titi || 3 -eq 3) ] && echo vrai
bash : syntax error near unexpected token ‘]’
% [[ 3 -eq 3 && (toto = titi || 3 -eq 3) ]] && echo vrai
vrai
```

5.4 Sous-programmes

- un script peut utiliser
  - un autre script stocké dans un autre fichier comme une commande standard ;
  - une fonction (définie dans le script lui-même ou ailleurs)

- les variables déclarées ou initialisées dans le programme sont locales ; ie invisibles par les processus engendrés

- ces variables peuvent être rendues visibles par la commande

```
export variables
```

- une fonction déclarée dans un script est inconnue des processus engendrés.

- fonction
  - macro-commande (définie par une séquence `()` ou {}, cf 3.9) à laquelle on donne un nom, et éventuellement des paramètres
  - syntaxe :

```
nom_fonction()
       macro_commande
```

  - paramètres : comme les paramèters du shell (`$1, $2` etc.), utilisables dans le corps de la fonction
  - exemple :

```
del()
 {
  rm $1
  echo "fichier $1 detruit"
 }
```



```
% del toto
fichier toto detruit
```

– exemple de sous programme :

script s1

```
#! /bin/ksh
rep=TOTO
export rep
s2
```

script s2

```
#!/bin/ksh
if [ -z $rep ]
   then
     rep=$PWD
fi
touch $rep/titi
```

→ s2 crée le fichier titi sous le répertoire courant si la variable rep est vide

→ s1 exécute s2 dans le répertoire TOTO : donc création du fichier TOTO/titi

→ quel est le résultat si on enlève la commande `export` ?

5.5 Optimiser un script

5.5.1 coût de création d'un processus

- mise à jour d'index et de tables
- coût fixe qui peut être important par rapport au coût du traitement des données
- coût négligeable si le processus gère beaucoup de données
- coût très important si peu de données (e.g dans une boucle)
- solution : utiliser des commandes qui travaillent sur un gros volume de données (commandes unix de traitement de textes : sed, awk, ...)

5.5.2 coût de création/destruction d'un fichier

- accès disque très coûteux
- mise à jour d'index et de tables
- coût important si le fichier contient peu de données
- solution : si peu de données, les stocker en mémoire dans une variable

5.5.3 optimisation

- attention au path
  - utiliser le nom absolu d'une commande unix fréquemment utilisée (car la recherche dans le path est longue)
  - le path ne doit pas être trop long
  - mettre le répertoire contenant les commandes les plus utilisées en début de path

- si une boucle est inévitable, préférer les commandes internes du shell et les opérations sur les variables au lieu des commandes unix (pas de nouveau processus). Exemple `[ ]` au lieu de `test`

- a contrario, limiter les lectures de fichier ligne à ligne (autant de processus que de lignes) en utilisant les commandes unix de traitement de texte comme sed ou awk. Ces commandes traitent un fichier avec un seul processus.

5.6 écrire un script propre

5.6.1 portabilité

script indépendant de la machine : si le script est écrit dans le langage SHELL, mettre en première ligne

```
#!/bin/SHELL
```

5.6.2 lisibilité

- commentaires `#`
- nom du programme, but, date de dernière modification, exemple d'utilisation complète avec les options

5.6.3 déchets

les fichiers temporaires ne doivent pas écraser des fichiers existants, et doivent être détruits en fin de programme

- où les mettre : dans `/tmp`
- nom à donner : nom au choix, terminé par le numéro du processus du programme `$$`, stocké dans une variable.

  exemple :

```
tmp1=/tmp/fich_temp$$
```

5.6.4 robustesse

le programme ne doit pas s'arrêter pour une raison inconnue

- test des options : en général un programme qui a plusieurs options possibles peut n'en accepter qu'une partie (voir les commandes usuelles comme `ls`)

  → présence :
  - nb d'options : correct ?
  - ordre : correct ?
  - option : connue ?

  → validité :
  - un nombre ou un caractère : valeur correcte ?
  - un fichier : existant ?

  Dans chaque cas, renvoyer une message d'erreur clair et la syntaxe à utiliser

– Traitement des signaux

  – comment demander confirmation à l'utilisateur après un CTRL-C ?
  – comment éviter la non-destruction d'un fichier temporararaire après l'interruption inattendue d'un programme ?
  – commande trap

```
trap commande liste_signaux
```

→ si le processus reçoit un des signaux de la liste_signaux, la commande est exécutée (le sens initial du signal est alors ignoré)

→ si la commande est la chaîne vide `""` les signaux de la liste sont simplement ignorés

→ si la commande ne contient pas l'instruction `exit`, le programme continue à s'exécuter à la ligne suivante

- principaux signaux (taper `kill -l` pour la liste complète) :
  - `1` ou `HUP` : signal envoyé en fin de session (déconnexion), utile pour éviter qu'un script tournant en tâche de fond soit arrêté à la déconnexion
  - `2` ou `INT` : interruption par `control c`
  - `9` ou `KILL` : interruption ultime

    (ne peut être dérouté par `trap`)

exemple :

```
#!/bin/ksh
# fonction d'arret
arret()
{
 echo "etes-vous sur ? (o/n)"
 read rep
 if [ $rep = o  ]
    then
        echo "$0 : arret" ; exit 0
    else
        echo "continue"
 fi
}
# attrape signaux
trap "arret" INT
```

```
# programme
sleep 1000
```

- messages d'erreur : les écrire dans la stderr, avec le nom du programme. Exemple :

```
echo "$0 : message_erreur" >&2
```

- code retour :
  - `exit n` en fin de programme force le code de retour à la valeur n
  - si on ne met rien, le code est celui de la dernière commande exécutée
  - intérêt : on peut considérer que l'exécution du programme est réussie même si la dernière commande a échouée
  - convention : code 0 pour un succès

6 Outils de transformations de textes

- but : traitement automatique de lignes de texte (recherche, remplacement, comptage, tri) par des commandes UNIX (interactive ou non), utilisable dans des programmes shell
- très utile pour traiter automatiquement de gros volumes de données (changement de format etc.)
- concept essentiel : expressions régulières

6.1 Expressions régulières

- but :
  - fabriquer des motifs qui correspondent à plusieurs chaînes de texte sur une seule ligne, pour des recherches/remplacements complexes
  - analogie : expressions génériques du shell permettent de construire des motifs correspondant à plusieurs noms de fichier
  - utilisation de caracères spéciaux, différents de ceux du shell, car les besoins sont différents (repérer le début de la ligne etc.)
  - exemple : trouver toutes les lignes d'un fichier fortran contenant l'appel à un module. Ces lignes commencent par `use` précédé d'un ou plusieurs blancs ou tabulations. Ceci n'est pas possibles avec une expression générique du shell.

– caractères spéciaux permanents

`.`	caractère quelconque	
`*`	le caractère qui le précède est répété 0 ou plusieurs fois	
	`a *`	vaut a suivit de 0 ou plusieurs blancs
`\`	le caractère spécial qui suit devient normal	
	`\*\**`	vaut * suivit de une ou plusieurs *
`[]`	un caractère parmi la liste donnée entre `[]`	
	`[abd9]`	a ou b ou d ou 9
	`[a-z]`	a ou b ou ... z
	`[a-z1]`	`[a-z]` ou 1
	`[ab][12]`	a1 ou a2 ou b1 ou b2
	`z[ab]z`	zaz ou zbz
	`[0-9][0-9]*`	tout nombre entier
`[^ ]`	un caractère hors de la liste donnée après `^`	

– caractères spéciaux de position

`^` → en début d'expression, indique qu'elle est en début de ligne

→ dans un `[]` voir ce qui précède

`^begin`	begin en début de ligne
`beg^in`	chaîne beg^in
`[^begin]`	tout caractère sauf b, e, g, i, n

`$` en fin d'expression, indique qu'elle est en fin de ligne

`end$`	end en fin de ligne
`[ab]$`	a ou b en fin de ligne
`$end`	$end
`^.*$`	une ligne entière quelconque
`^$`	la ligne vide

– notations des caractères TAB, CTRL etc.

voir la documentation texinfo sur les expressions régulières donnée pour la commande grep (taper **`info grep`**)

exemple en **`bash`** : caractère blanc ou TAB

```
[:blank:]
```

– dangers : le principal est la phase d'interprétation par le shell d'une commande utilisant une expression régulière. Quand c'est possible, le plus sûr est d'encadrer cette expression par des quotes `'`

exemple précédent :

– afficher toutes les lignes d'appel à un module fortran
– l'er est `^[[:blank:]]*use.*$` (ligne commençant par 0 ou plusieurs blancs ou tabulations, suivis de use, suivi de 0 ou plusieurs caractères quelconques, jusqu'à la fin de la ligne)
– si on utilise la commande **`grep`** ainsi (cf page suivante)

```
grep ^[[:blank:]]*use.*$ prog.f90
```

ça ne marche pas car après interprétation, le 1er argument fourni à **`grep`** est la chaîne `^[`, car le blanc est un séparateur de champ
– avec des quotes, ça marche

```
grep '^[[:blank:]]*use.*$' prog.f90
```

– extensions d'expressions régulières : soit c un caractère éventuellement donné par une e.r

`c\{m,n\}`	vaut c répété entre m et n fois
`c\{m,\}`	vaut c répété au moins m fois
`c\{m\}`	vaut c répété exactement m fois

exemple :

`za\{2,4\}`	zaaz ou zaaaz ou zaaaaz
`[to]\{2\}`	tt ou to ou ot ou oo
`^[a-z]\{10\}$`	une ligne contenant uniquement 10 minuscules

6.2 Recherche de chaîne : commande grep

recherche d'une chaîne de caractères dans un ou plusieurs fichiers :

– si motif est une expression régulière :

`grep` *options* `motif liste_fichiers`

– si motif est une chaîne ordinaire :

`egrep` *options* `motif liste_fichiers`

– sans option : affiche les lignes du fichier contenant le motif. Dans le cas d'une liste de fichiers, le nom des fichiers est aussi affiché

– option utile : `-i` ignore la distinction minuscule/majuscule

exemples :

– rechercher les créations de matrices creuses (*sparse*) du programme matlab *prog.m* :

```
grep sparse prog.m
```

– affichage de la liste des fichiers du répertoire personnel avec les droits suivants pour l'utilisateur : accessibles ou pas en lecture, inaccessibles en écriture, et accessibles en exécution :

```
ls -l | grep '^-[r-]-x'
```

6.3 Modification de texte : éditeur sed

- permet de traiter automatiquement une suite de caractères pour toutes les lignes d'un fichier : suppression, copie, remplacement. Pas besoin de visualiser le fichier avec un éditeur classique (emacs, etc.)

- syntaxe :

  `sed` *options* `-e` *requete1* `-e` *requete2 ... fichier*

- une requête est une commande (remplacement, destruction, etc.) éventuellement précédée des adresses des lignes à traiter

- si on ne donne pas de nom de fichier, l'entrée est lue au clavier

- par défaut, le résultat de cette commande est envoyé à l'écran. Il faut donc une redirection pour en faire un fichier

– adresses des lignes à traiter

`n`	ligne n
`m,n`	les lignes m à n
`n,$`	la ligne n à la dernière
`/er/`	lignes contenant l'e.r
`+er+`	idem, encadrement avec un autre signe que / si l'e.r contient ce caractère (ici +)
`/er1/,/er2/`	lignes des blocs commençant par une ligne contenant l'er1 et finissant par une ligne contenant l'er2

si on ne donne pas d'adresse, toutes les lignes sont traitées

– exemples :

`128`	ligne 128
`128,241`	les lignes 128 à 241
`128,$`	la ligne 128 à la dernière
`/^[^0-9]*$/`	lignes ne contenant aucun chiffre
`/begin/,/end/`	lignes des blocs commençant par une ligne contenant la chaîne begin et finissant par une ligne contenant la chaîne end
`/prog/,$`	toutes les lignes comprises entre la première contenant la chaîne prog et la dernière du fichier

– commande de substitution

```
s/er_a_remplacer/chaîne_de_remplacement/g
```

s commande

/ séparateur. On peut utiliser n'importe quel caractère, mais il doit suivre immédiatement la commande **s**

g optionnel, toutes les occurences de l'er sont remplacées (la première sinon)

exemple :

```
sed -e '/^[Cc].*$/ s/[Cc]/!/' prog.f
```

le caractère de commentaire c (fortran 77) du fichier prog.f est remplacé par le caractère de commentaire ! (f90)

– commandes de suppression : `d`

les lignes dont les adresses sont fournies dans le 1er argument de sed sont supprimées. Pratiquement, les lignes non supprimées sont envoyées à l'écran.

exemple :

`sed -e '/begin{figure}/,/end{figure}/ d' rapport.tex`

supprime tous les blocs définissant une figure dans le fichier LaTeX rapport.tex

– commande de duplication : `p`

fonctionne de la même façon que la destruction.

– commande `p` avec option `-n` : seules les lignes sélectionnées sont envoyées sur stdout.

exemple :

```
sed -n -e '/^[[:blank:]]*integer/ p' prog.f90
```

toutes les déclarations d'entiers du programme prog.f90 sont affichées à l'écran

– substitution avec tampon (*buffer*) : la chaîne de remplacement contient tout ou partie de la chaîne à remplacer

exemple : remplacer dans un texte tous les nombres négatifs par leur valeur absolue (i.e enlever le signe −)

→ tampon principal : toute la chaîne à remplacer est nommée par `&` dans la chaîne de remplacement

exemple : rajouter un ; en fin de ligne à toutes les lignes contenant une affectation de variables MAPLE, sauf s'il y a déjà un ; ou un :

```
sed -e 's/:=.*[^;:]$/&;/' toto.mws
```

→ tampons secondaires :

- pour utiliser une partie de la chaîne à remplacer : on l'encadre dans la chaîne par des `\(   \)`
- on la désigne par `\1` dans la chaîne de remplacement
- si on encadre d'autres parties de la même façon, elles seront désignés par `\2`, `\3` etc.

exemple 1 : tous les entiers négatifs sont remplacés par leur valeur absolue

```
sed -e 's/-\([0-9][0-9]*\)/\1/g'
```

exemple 2 : inverser les indices
des éléments du type A[i,j] de la matrice A dans un fichier MAPLE

```
sed -e 's/A\[\([^,][^,]*\),\([^,][^,]*\)\]/A\[\2,\1\]/g'
```

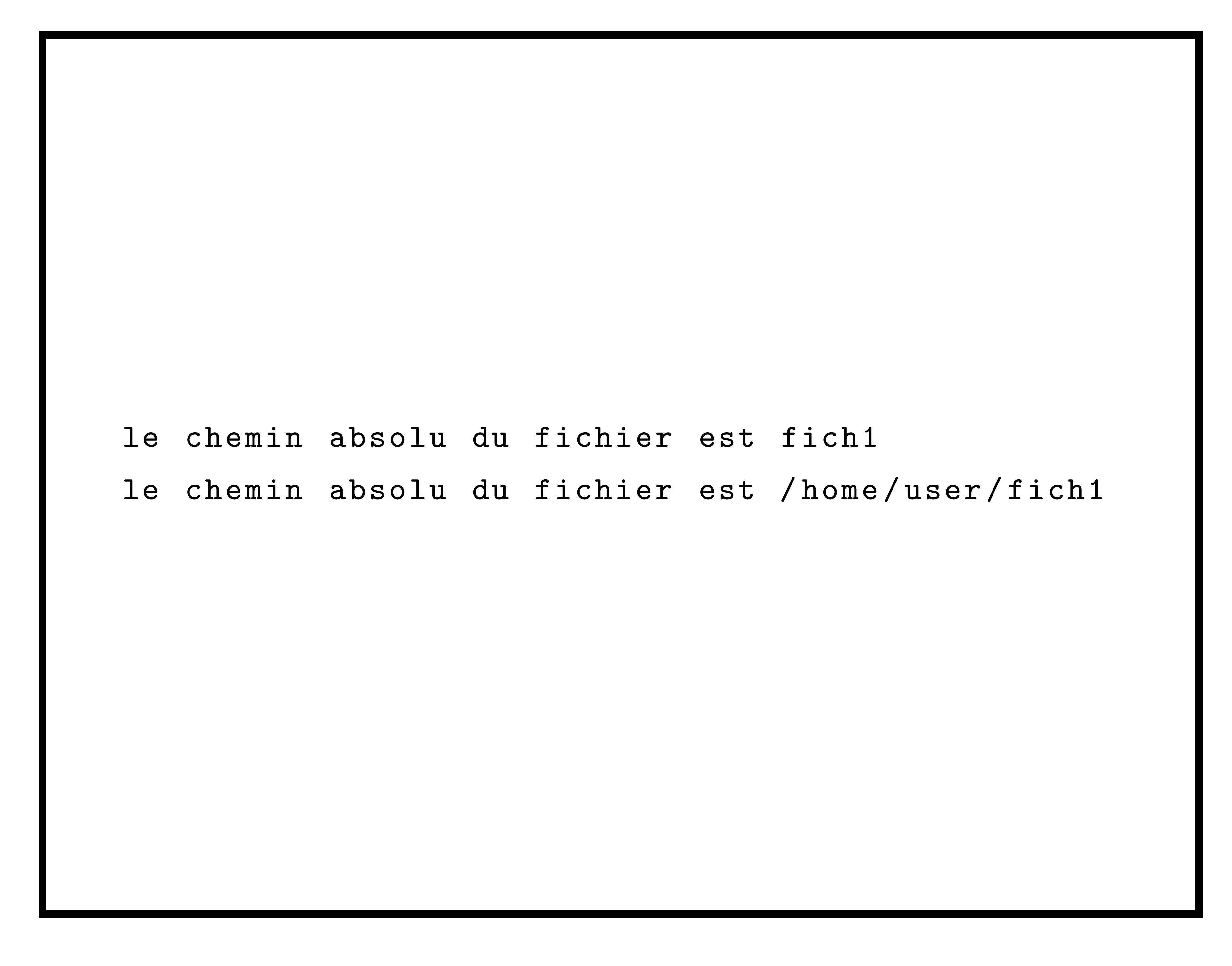
- pour travailler sur des champs plutôt que sur des suites de caractères (c.-à-d. des chaînes séparées par des séparateurs), il vaut mieux utiliser `awk`

- ATTENTION : en général on encadre chaque requête de sed (adresses et commande) entre quotes `'`, pour éviter le risque d'interprétation des caractères spéciaux par le shell

- si on veut utiliser des contenus de variables du shell, il vaut alors mieux utiliser des doubles quotes `"`

  exemple :

```
% echo $PWD
/home/user
% sed -e "s+fich1+$PWD/&+"
```


le chemin absolu du fichier est fich1
le chemin absolu du fichier est /home/user/fich1

6.4 traitement de texte : éditeur awk

– utilisation basique :

`awk` *selection* {`action`} *fichier*

→ *selection* permet de sélectionner des lignes du fichier, par exemple

`/er/`	lignes contenant l'er
`/er1/er2/`	bloc compris entre la ligne contenant er1 et la ligne contenant er2

→ {`action`} peut par exemple être un affichage

{`print chaîne`}

→ les champs des lignes sont notés `$1`, `$2`, etc. et sont utilisables dans la selection et l'action

- exemple : afficher le mois et le nom de chaque sous répertoire du répertoire courant

```
ls -l | awk '/^d/{print $6, $9}'
```

- awk peut être utilisé de façon beaucoup plus poussée :

  - on peut spécifier le caractère séparateur des champs
  - il existe d'autres variables que les numéros de champs
  - les actions peuvent être des affectations de variables, des tests if, des boucles for ou while, des calculs
  - plusieurs actions peuvent être données dans un fichier appelé programme awk
  - les sélections peuvent utiliser des opérateurs booléens et des variables
  - pour seulement afficher certains champs sans selection de ligne, la commande unix `cut` peut suffire

6.5 autres commandes

- **`paste`** : fusionne deux ou plusieurs fichiers ligne à ligne
- **`cut`** : extraire des colonnes d'un texte
- **`tr`** *option* **`chaîne1`** *chaîne2* : substitue les caractères de chaîne1 à ceux de chaîne2. Si chaîne2 est vide, les caractères de chaîne1 sont détruits. L'entrée standard est le clavier, la sortie standard est l'écran. Les chaînes ne sont pas des expressions régulières, mais des notations spéciales sont acceptées, comme par exemple

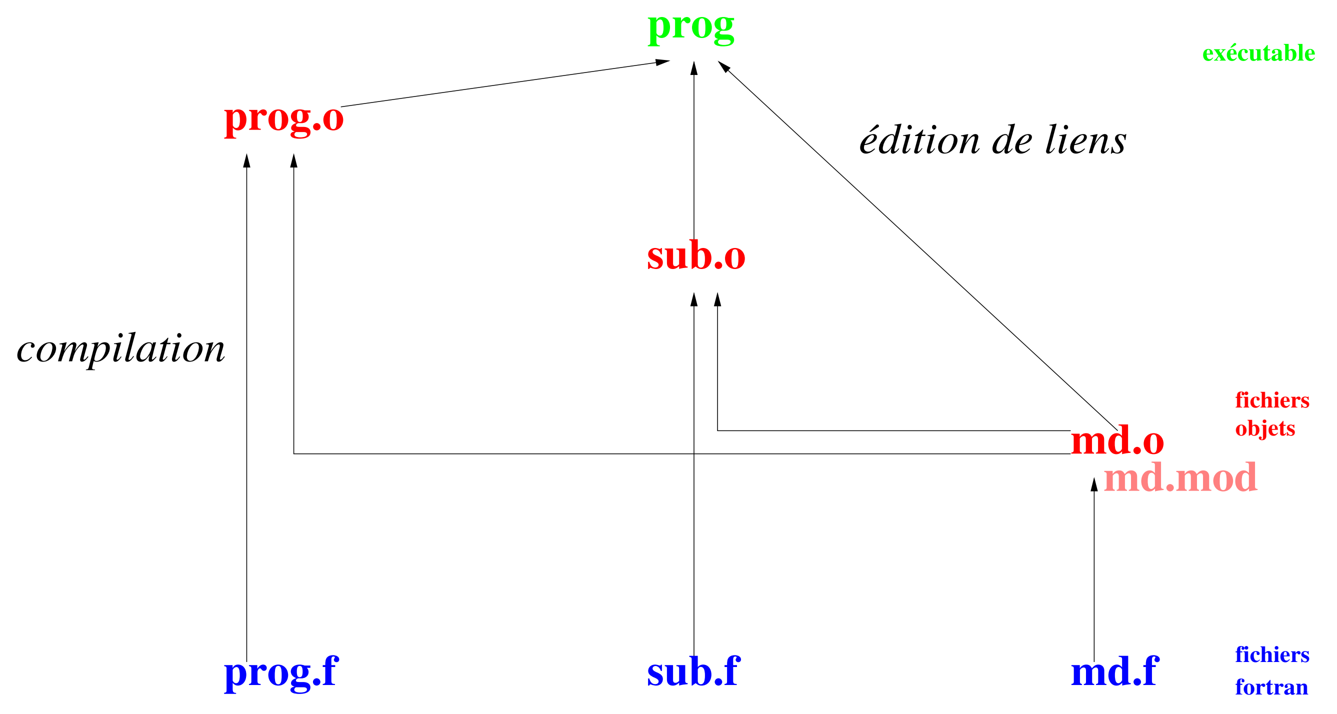
  ```
  tr [a-z] [A-Z]
  ```

  qui remplace les minuscules par des majuscules

7 utilisation de la commande make pour la compilation séparée

- logiciel qui suit un mode d'emploi (*makefile*) pour fabriquer des fichiers dépendant les uns des autres
- utile pour la compilation séparée

– exemple : programme fortran90 composé d'un programme principal `prog` et d'une subroutine externe `sub`. Programme et subroutine utilisent le module `md`.



→ la modification de sub.f90 ne nécessite pas la recompilation de prog.f90 et mod.f90

→ seules la recompilation de sub.f90 et l'édition de liens sont nécessaires

– principe :

→ le fichier mode d'emploi indique les dépendances des fichiers et l'action à exécuter pour construire chaque fichier (compilation fortan ou c++, latex etc.)

→ make utilise les dates des fichiers pour vérifier s'ils sont à jour ou pas en suivant les règles de dépendances données dans le mode d'emploi

exemple : il faut recompiler sub.f90 et refaire l'édition de liens si la date de sub.f90 est postérieure à celle de prog.

– écriture d'un mode d'emploi

→ en général stocké dans un fichier nommé *makefile*

→ ligne de dépendances :

fichier-à-faire `:` *dépendance1 dépendance2 ...*

→ lignes actions à exécuter (sous la ligne de dépendance)

```
<TAB> action1
<TAB> @ action2 (le @ pour éviter l'affichage
                 de la ligne de commande)
```

→ ligne règle implicite :

```
.s.t :
<TAB> action
```

→ définition de macro-commande :

nom-macro `=` *définition-macro*

→ ligne de commentaires :

`#` *commentaires*

– exemple de mode d'emploi :

```
# mode d'emploi compilation programme prog
# edition de lien
prog : md.o sub.o prog.o
<TAB>  f90 -o prog prog.o sub.o md.o

# compilation
prog.o : prog.f90 md.mod
<TAB> f90 -c prog.f90

sub.o : sub.f90 md.mod
<TAB> f90 -c sub.f90

md.o md.mod : md.f90
<TAB> f90 -c md.f90
```

→ la compilation du programme peut alors être effectuée par make :

```
make   ou   make prog
make sub.o
make md.o
```

→ si le fichier à faire est à jour, make affiche :

fichier `is up to date`

- ATTENTION :
  - → par défaut, make va essayer de faire la première cible ;
  - → il ne va voir les autres cibles que si ce sont des dépendances de la première ;
  - → les cibles non nécessaires pour la première ne sont pas traitées ;
  - → par conséquent il faut toujours mettre le fichier final (l'exécutable) en premier dans le mode d'emploi.

- exemple :
  - → dans l'exemple précédent, mettre la règle de la cible prog à la fin du fichier ;
  - → modifier le module md, et lancer make ;
  - → on constate alors que la subroutine sub n'est pas recompilée, ce qui n'est pas correct.

– les macro-commandes : une fois définies au début du mode d'emploi, elles sont utilisables par l'expression

```
$(nom-macro)
```

→ permet d'avoir un nom raccourci pour une commande complexe et répétée

→ peut servir à paramétrer un mode d'emploi, en initialisant la macro-commande dans l'appel de make :

```
make nom-macro=definition
```

cette initialisation annule celle écrite dans le mode d'emploi

– exemple : pour compiler le programme précédent en mode debug (option -g) ou en différents modes optimisés (options -O1, -O2, -O3)

→ on crée les macros vides en début de mode d'emploi

```
DEBUG=
OPT=
```

→ on remplace les actions `f90` par

```
f90 $(DEBUG) $(OPT)
```

→ pour compiler en mode debug, on appelle make ainsi :

```
make DEBUG=-g
```

→ pour compiler en mode optimisé 3, on appelle make ainsi :

```
make OPT=-O3
```

→ pour compiler en mode standard, on appelle make ainsi :

```
make
```

– macros prédéfinies :

`$@`	nom du fichier à faire
`$*`	nom du fichier à faire sans suffixe
`$<`	nom du fichier prérequis qui provoque l'exécution de l'action

– caractère de continuation de ligne : `\`

exemple :

```
toto.o : toto1.f90 toto2.f90  \
 toto3.f90 toto4.f90
```

– règle implicite : permet de ne pas préciser d'action quand deux fichiers sont liés par une telle règle

exemple : la règle implicite pour fabriquer un fichier .o à partir d'un fichier .f est la compilation avec f77. Si on veut la modifier pour la remplacer par la compilation f90, on écrit au début du mode d'emploi

```
.f.o :
<TAB> f90 -o $@ $<
```

et on rajoute ces deux suffixes dans la liste `.SUFFIXES` au début du mode d'emploi

```
.SUFFIXES : .f .o
```

– règle sans fichier (*phony target*) :

```
nom :
<TAB> action
```

si nom n'est pas un fichier, l'action sera exécutée avec `make` *nom*

– exemple final : mode d'emploi précédent

```
.SUFFIXES :
.SUFFIXES : .f90 .o .mod
DEBUG=
OPT=
COMPILE=f90 $(DEBUG) $(OPT)

# edition de liens
prog : prog.o sub.o md.o
<TAB> $(COMPILE) -o prog prog.o sub.o md.o
<TAB> @echo compilation terminée

# compilation
prog.o : prog.f90 md.mod
<TAB> $(COMPILE) -c prog.f90
sub.o : sub.f90 md.mod
<TAB> $(COMPILE) -c sub.f90

# regles implicites
.f90.o :
<TAB> $(COMPILE) -c $<
.f90.mod :
<TAB> $(COMPILE) -c $<

# destruction des fichiers objets et modules
detruire :
<TAB> rm -f *.o *.mod
```

– options

`-n`	commandes affichées, pas exécutées (pour la mise au point)
`-i`	erreurs ignorées (sinon arrêt)
`-r`	liste de règles implicites ignorée
`-d`	tout s'affiche (utile pour comprendre le fonctionnement de make)
`-f` *fich*	pour donner un autre fichier que celui par défaut (makefile)

- à noter : le make de GNU améliore la fabrication de règles implicites avec les règles de motif (*pattern rules*) (voir la doc. de gnumake [2])
- il existe des logiciels qui fabriquent automatiquement des makefiles. Ceci peut être intéressant pour les gros codes , en particulier en fortran 90 où l'ordre de compilation des modules est primordial (voir par exemple le script `fmkmf` sur `http://www.met.ed.ac.uk/~hcp/fmkmf.html`)
- un nouveau logiciel (`cons`) est plus puissant que `make`, voir sur le site de GNU. En particulier, avec `cons`, la modification du corps d'un module sans modification de son interface n'entraîne pas la recompilation des fichiers dépendants.

Références

[1] MANUELS GNU.
`http ://www.gnu.org/software/bash/manual/bash.html`.

[2] MANUELS GNU.
`http ://www.gnu.org/software/make/manual/make.html`.

[3] MANUELS GNU.
`http ://www.gnu.org/software/sed/manual/sed.html`.

[4] J. L. Nebut. *UNIX pour l'utilisateur - Commandes et Langages de commandes*. Éditions Technip, 1990.

[5] J.-F. Pujol. Guide du korn-shell sous unix.
`http ://www-ensimag.imag.fr/cours/Systeme /documents/shell/Korn.Shell.pdf`.